Applications of Storage Mapping Optimization to Register Promotion

Patrick Carribault
Bull, Les Clayes-sous-Bois
PRiSM, Université de Versailles

Albert Cohen
ALCHEMY Group
INRIA Futurs and LRI, Université Paris-Sud
HiPEAC Network

ABSTRACT

Storage mapping optimization is a flexible approach to folding array dimensions in numerical codes. It is designed to reduce the memory footprint after a wide spectrum of loop transformations, whether based on uniform dependence vectors or more expressive polyhedral abstractions. Conversely, few loop transformations have been proposed to facilitate register promotion, namely loop fusion, unroll-and-jam or tiling. Building on array data-flow analysis and expansion, we extend storage mapping optimization to improve opportunities for register promotion.

Our work is motivated by the empirical study of a computational biology benchmark, the approximate string matching algorithm BPR from NR-grep, on a wide issue micro-architecture. Our experiments confirm the major benefit of register tiling (even on non-numerical benchmarks) but also shed the light on two novel issues: prior array expansion may be necessary to enable loop transformations that finally authorize profitable register promotion, and more advanced scheduling techniques (beyond tiling and unroll-and-jam) may significantly improve performance in fine-tuning register usage and instruction-level parallelism.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Optimization.*

General Terms

Algorithms, Performance.

Keywords

Register Promotion, Tiling, Blocking, Scheduling, Array Contraction, Array Folding, Pattern Matching, String Matching, Itanium.

1. INTRODUCTION

Scalar optimizations applied to register promotion are ubiquitous in production compilers and successful in improving performance of a large spectrum of programs. Most of them are based on constant sub-expression elimination or partial redundancy elimination, combined with pointer analysis and loop unrolling [34, 31, 7].

ICS'04, June 26–July 1, 2004, Saint-Malo, France.

Register promotion for large register files must be combined with more aggressive loop transformations. First of all, it is critical to exploit scalar reuse over iterations of the innermost loop [16], e.g., to optimize the benefits of software pipelining. Register tiling is a complementary approach to exhibit scalar reuse at different depths in a loop nest; it plays the first role in optimization strategies for many memory-bound kernels from numerical, multimedia, cryptography, or computational biology benchmarks. A typical example is matrix-matrix multiplication [36], it is also the case for the computational biology example described in this paper. Most works build on locality-improving techniques like loop fusion [8, 10, 33] or tiling [11, 33, 21] to exhibit more opportunities for scalar promotion. In this paper, we will refer to both tiling and unroll-and-jam based techniques as *register blocking*. Recent improvements to these approaches proved their applicability to wide issue machines [21] and codesign and synthesis tools [42, 43].

1.1 Running Example

We will now present a simple example to clarify the basic register blocking concepts and to motivate the use of array expansion and storage mapping optimization. The nest in Figure 1 is typical of hand-optimized programs with scalar reuse, both intra-loop — `n` and `p` — and loop-carried — `n` and `o`. Dependences on `D` (and scalars) hamper register blocking: *it does not seem possible to further reduce the number of memory accesses*.

Dependence removal

To enable unroll-and-jam or tiling, one may first convert `D` to the single-assignment array `E` in Figure 2, thanks to array data-flow analysis [17], then eliminate scalars `n`, `o` and `p` by forward substitution, see Figure 3. The expansion of `D` removes all memory-based dependences on arrays, while forward substitution removes output and anti dependences on scalars `n` and `o` (at the cost of redundant memory accesses, but at a lower cost than expanding scalars `n` and `o` into arrays). Notice other dependence removal techniques are not really applicable to this case: array renaming is not applicable [24], and array privatization [46] does not easily apply in the presence of loop-carried dependences and induces a copy overhead that may only be fully eliminated by array data-flow analysis, the core technique for conversion to single assignment form. One may now unroll the outer loop, fuse the resulting inner loops, and finally unroll the inner loop, see Figures 4 and 5 (we use unroll factors of 2 and assume m and k are odd numbers for the sake of clarity). Figure 7 shows array data-flow dependences (arrows) and the resulting load (L) and store (S) pattern on the 2×2 (fully unrolled) tile. A simple scalar promotion algorithm [34] will only take intra-loop reuse into account and lead to the code in Figure 6. The corresponding load/store pattern is shown in Figure 8.

```
for (i=0; i<m; i++)
  o = n = D[0];
  for (j=0; j<k; j++)
    p = D[j];
    n = f(n, o, p);
    D[j] = n;
    o = p;
```

1 load and 1 store per iteration of j.

Figure 1: Original nest

```
for (i=0; i<m; i++)
  o = n = D[0];
  for (j=0; j<k; j++)
    if (i==0) p = D[0];
    else p = E[i-1][j];
    n = f(n, o, p);
    o = p;
    E[i][j] = n;
```

Figure 2: Expansion

```
E[0][0] = D[0];
for (j=1; j<k; j++)
  E[0][j] = f(E[0][j-1], D[0], D[0]);
for (i=1; i<m; i++)
  E[i][0] = f(D[0], D[0], E[i-1][0]);
  for (j=1; j<k; j++)
    E[i][j] = f(E[i][j-1], E[i-1][j-1], E[i-1][j]);
```

Figure 3: Scalar forward substitution and peeling

```
// first iteration omitted
for (i=1; i<m; i+=2)
  E[i][0] = f(D[0], D[0], E[i-1][0]);
  E[i+1][0] = f(D[0], D[0], E[i][0]);
  for (j=1; j<k; j++)
    E[i][j] = f(E[i][j-1], E[i-1][j-1], E[i-1][j]);
    E[i+1][j] = f(E[i+1][j-1], E[i][j-1], E[i][j]);
```

Figure 4: Unroll-and-jam

```
// first iteration omitted
for (i=1; i<m; i+=2)
  E[i][0] = f(D[0], D[0], E[i-1][0]);
  E[i+1][0] = f(D[0], D[0], E[i][0]);
  for (j=1; j<k; j+=2)
    E[i][j] = f(E[i][j-1], E[i-1][j-1], E[i-1][j]);
    E[i+1][j] = f(E[i+1][j-1], E[i][j-1], E[i][j]);
    E[i][j+1] = f(E[i][j], E[i-1][j], E[i-1][j+1]);
    E[i+1][j+1] = f(E[i+1][j], E[i][j], E[i][j+1]);
```

Figure 5: Inner loop unrolling

```
// first iteration omitted
d0 = D[0];
for (i=1; i<m; i+=2)
  e30 = f(d0, d0, E[i-1][0]);
  E[i][0] = e30;
  E[i+1][0] = f(d0, d0, e30);
  for (j=1; j<k; j+=2)
    e11 = E[i-1][j];
    e12 = E[i][j-1];
    e10 = f(e12, E[i-1][j-1], e11);
    e20 = f(E[i+1][j-1], e12, e10);
    e30 = f(e11, e10, E[i-1][j+1]);
    E[i+1][j] = e20;
    E[i][j+1] = e30
    E[i+1][j+1] = f(e20, e10, e30);
```

0.5 load and 1 store for $j = 0$.

1.25 load and 0.75 store per subsequent iteration of j.

Figure 6: State-of-the-art scalar promotion

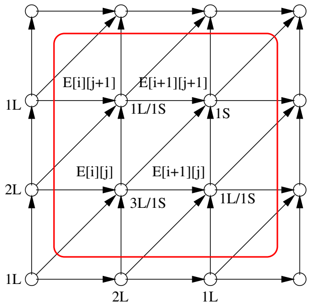


Figure 7: Flow of Figure 5

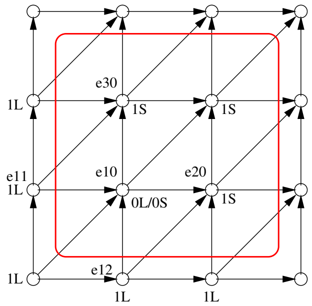


Figure 8: Flow of Figure 6

Overall, *array expansion seems a good idea to reduce the impact of manual optimizations on the applicability of important compiler phases*. Yet the result is worse than the original code: one spare store is traded for an additional load every four iterations of j, and the array is now two-dimensional. The additional load comes from the lack of scalars reuse along the inner loop (vertical dependence): the hand-optimized version dedicated n to the value that is now loaded from E[i][j-1]. In addition, no dimension of E can be eliminated by array contraction [38], since live values flow from both leftward and downward tiles. Such a disappointing result ruins the applicability of array expansion as an enabling technique for register blocking. Of course, this does not improve when applying more advanced rescheduling transformations before scalar promotion (beyond unroll-and-jam).

Notice that some extended scalar promotion techniques discover scalar reuse across loop iterations, most notably the one by Duesterwald et al. [16], but along the *innermost* loop only. Compared to what can be done with array data-flow analysis, these techniques fail to discover many cases of scalar reuse because they rely on simpler (and faster) analyses of array references. They typically fail for most imperfectly nested loops (including the example in Figure 6), for tiled nests where the innermost loops have not been fully unrolled, and for complex loop bounds and dependence patterns, like the skewed version of our benchmark example in the next section. In any case, such techniques would not succeed in folding the expanded arrays.

Exploring a more ambitious approach

We face two challenges:

1. enable loop-carried and cross-loop scalar reuse at every level of an imperfect loop nest, in the context of complex (but regular) array data-flow information;
2. guarantee that the memory footprint will be minimal, undoing any unnecessary array expansion induced by the conversion to single-assignment form.

To address the first issue, we duplicate the declaration of array E for each surrounding loop. The new arrays are called *reuse buffers*: they store *every value that will be reused across the corresponding loop iterations*. Practically, right after unroll-and-jam, we duplicate writes to E[i+1][*] — values flowing to the next iteration of the outer loop — with references to a new array B, and we replace reads from E[i-1][*] — values flowing from the previous iteration of the outer loop — by references to B, see Figure 9. After unrolling the inner loop, we repeat the process and store the values reused along the inner loop in a new array C, see Figure 10. This technique generalizes the inter-iteration reuse mechanism of [16] and extends the virtual register concept [31] to arrays. These progresses make *inter-iteration and cross-loop reuse applicable to a large class of imperfectly nested loops and arbitrary loop levels.*

Next, we may reduce the memory footprint with array folding techniques. The second dimension in array C is easily removed by array contraction, but arrays B and E require a finer-grain folding technique. This is precisely what storage mapping optimization is intended for [26]. Studying the number of intermediate writes during the lifetime of values produce at every iteration, one may

```
// first iteration omitted
for (i=1; i<m; i+=2)
  E[i][0] = f(D[0], D[0], B[i-1][0]);
  E[i+1][0] = f(D[0], D[0], E[i][0]);
  B[i+1][0] = E[i+1][0];
  for (j=1; j<k; j++)
    E[i][j] = f(E[i][j-1], B[i-1][j-1], B[i-1][j]);
    E[i+1][j] = f(E[i+1][j-1], E[i][j-1], E[i][j]);
    B[i+1][j] = E[i+1][j];
```

Figure 9: Outer loop buffer

```
// first iteration omitted
for (i=1; i<m; i+=2)
  E[i][0] = f(D[0], D[0], B[i-1][0]);
  E[i+1][0] = f(D[0], D[0], E[i][0]);
  C[i][0] = E[i][0];
  C[i+1][0] = E[i+1][0];
  B[i+1][0] = E[i+1][0];
  for (j=1; j<k; j+=2)
    E[i][j] = f(C[i][j-1], B[i-1][j-1], B[i-1][j]);
    E[i][j+1] = f(E[i][j], B[i-1][j], B[i-1][j+1]);
    E[i+1][j] = f(C[i+1][j-1], C[i][j-1], E[i][j]);
    E[i+1][j+1] = f(E[i+1][j], E[i][j], E[i][j+1]);
    C[i][j+1] = E[i][j+1];
    C[i+1][j+1] = E[i+1][j+1];
    B[i+1][j] = E[i+1][j];
    B[i+1][j+1] = E[i+1][j+1];
```

Figure 10: Inner loop buffer

check that only two columns of B (separated by a useless column recording no value) and two lines and two columns of E are simultaneously alive. Analogously, one may improve the folding of array C since only two of its elements are simulataneously alive. Using the technique by Lefebvre and Feautrier, we obtain the code in Figure 11 (the integer division in the subscript of B is due to the removal of the useless column).[1] The corresponding load/store pattern is shown in Figure 13.

The introduction of buffers B and C split the iteration space of values produced and consumed within a tile — array E — from the iteration space of values flowing to the upward tile — array C — and from the iteration space of values flowing to rightward tiles — array B. Storage mapping optimization exploits this iteration-space partitioning to decouple the folding of each array, B, C and E, hence to further reduce the memory footprint, and to discover opportunities for scalar promotion: it is now straightforward to promote the small, bounded size arrays C and E to scalars. Applying intra-block scalar promotion to B as well, we obtain the optimized code in Figure 12. The corresponding load/store pattern is shown in Figure 14. This final version needs only three fourths of the loads and half the stores of the original code. In addition, considering the peeled iteration $j = 0$, scalar promotion eliminates all array references but one, reusing values of C when entering the inner loop and reusing values of B across iterations of the outer loop. This kind of scalar reuse is out of reach of previously proposed techniques.

1.2 Facilitating Register Promotion

Previous works studied loop transformations facilitating register promotion (mostly) in isolation from the other optimization phases. A real loop nest optimizer includes transformations to deal with the cache hierarchy and to exploit instruction or thread level par-

[1] Their method generates modulo operations for arrays C and E as well, of the form C[i%2] and E[i%2][j%2], but these subscripts can be further simplified because i and j are odd numbers. Likewise, notice that the complex subscript of array B may be simplified by further unrolling (or strip-mining) the outer loop.

```
// first iteration omitted
for (i=1; i<m; i+=2)
  E[1][0] = f(D[0], D[0], B[(i/2)%2][0]);
  E[0][0] = f(D[0], D[0], E[1][0]);
  C[1] = E[1][0];
  C[0] = E[0][0];
  B[(i/2+1)%2][0] = E[0][0];
  for (j=1; j<k; j+=2)
    E[1][1] = f(C[1], B[(i/2)%2][j-1], B[(i/2)%2][j]);
    E[1][0] = f(e10, B[(i/2)%2][j], B[(i/2)%2][j+1]);
    E[0][1] = f(C[0], C[1], E[1][1]);
    E[0][0] = f(E[0][1], E[1][1], E[1][0]);
    C[1] = E[1][0];
    C[0] = E[0][0];
    B[(i/2+1)%2][j] = E[0][1];
    B[(i/2+1)%2][j+1] = E[0][0];
```

Figure 11: Array folding

```
// first iteration omitted
d0 = D[0];
for (i=1; i<m; i+=2)
  e30 = f(d0, d0, e40);
  e40 = f(d0, d0, e30);
  c1 = e30; // can be
  c2 = e40; // forwarded
  B[(i/2+1)%2][0] = e40;
  for (j=1; j<k; j+=2)
    b = B[(i/2)%2][j];
    e10 = f(c1, B[(i/2)%2][j-1], b);
    e30 = f(e10, b, B[(i/2)%2][j+1]);
    e20 = f(c2, c1, e10);
    e40 = f(e20, e10, e30);
    c1 = e30; // can be
    c2 = e40; // forwarded
    B[(i/2+1)%2)/2][j] = e20;
    B[(i/2+1)%2)/2][j+1] = e40;
```

0 load and 0.5 store for $j = 0$.

0.75 load and 0.5 store per subsequent iteration of j.

Figure 12: Improved scalar promotion

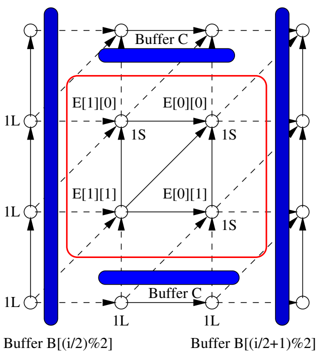


Figure 13: Flow of Figure 10

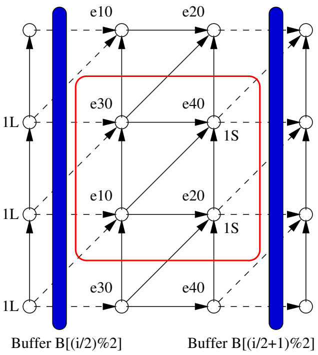


Figure 14: Flow of Figure 12

allelism. Unfortunately, controlling the interplay of loop transformations is one of the hardest problem for todays loop-restructuring compilers: first of all, predicting the effect of loop transformations is tough on micro-architectures with complex dynamic structures like branch predictors and reorder buffers [20], and predicting their interactions is even worse [36]; in addition, classical transformations do not compose easily because they rely on fragile pattern matching rules [12].

Of course, no technique is supposed to solve all performance problems, but a practical loop transformation for register promotion should minimize the constraints imposed on other transformation phases. For example, loop distribution or skewing [1] may favor software pipelining, exposing more ILP without degrading instruction cache locality. Register promotion should not forcibly tile loops and rule out other important transformations. Our work addresses this interplay issue by three means:

- decoupling the array folding technique from the scheduling algorithm or profitability heuristic for loop transformation;
- decoupling the array folding technique from unrolling and low-level register handling mechanisms (e.g., rotation);
- relying on storage-mapping optimization [26, 44, 40, 45], a generalized approach to array folding compatible with the most advanced scheduling algorithms.

We describe a polyhedral approach combining array expansion with affine scheduling and storage mapping optimization.

Array expansion

Unlike other array expansion techniques, conversion to (dynamic) *single-assignment* form [17] removes *every* memory-based (output and anti) dependence. This improves the potential for loop transformations and reduces the impact of syntactic variations in the indexing scheme of the original arrays. Maximal static expansion [2] is a natural extension suitable for irregular nests: it removes a memory dependence only when the producer of a value is statically known (i.e., without evaluating ϕ functions at run-time).

At first glance, array expansion seems the exact opposite of our final goal. However, removing spurious dependence constraints favors the combination of register promotion with other optimizations. Interestingly, it also favors register promotion itself by enabling alternative array folding opportunities with better impact on memory traffic. In addition, we will show that storage mapping optimization controls the overhead of array expansion on the remaining memory accesses.

Affine scheduling

Most schedule-oriented loop optimizations can be modeled with affine schedules. It is the case for unimodular transformations, loop fusion, unroll-and-jam, tiling, software pipelining and statement reordering [22, 27, 5], as well as more complex transformations like shackling [25] or chunking [6]. In addition, several recent locality-improving heuristics build directly on affine schedules [45, 29, 6]. The abstraction level of this model allows for register reuse across iterations of any loop and across different loops, without suffering from nesting restrictions.

Storage mapping optimization

Designed to reduce the memory footprint after a wide spectrum of loop transformations, storage mapping optimization is a flexible generalization of array contraction [28, 38]. It supports both uniform dependence vectors [9, 44] and polyhedral abstractions [26, 40, 45]. The last three techniques benefit from array data-flow analysis [17, 39, 14, 47, 3] and affine scheduling [19] to discover and exploit liveness information.

Our work extends the algorithm by Lefebvre and Feautrier [26]. It can be generalized to irregular loop nests along the lines of [13], and this is the main reason why we prefer this technique to the algorithm by Quilleré and Rajopadhye [40]. However, we would not be able to directly apply the techniques by Thies et al. [45] because it cannot contract more than one array dimension, which is required to reduce memory footprint after conversion to single-assignment form.

Eventually, whereas traditional storage mapping folds multiple array elements to a single location to reduce the memory footprint, our technique combines this effect with buffer insertion (or index-set splitting) to decrease the memory access rate in a loop nest, and to factor memory transfers to folded locations.

1.3 Applications and Contributions

Our work is motivated by the empirical study of a computational biology benchmark — the approximate pattern matching algorithm BPR from NR-grep [35] — on a wide issue architecture. Our experiments confirm the major benefit of register blocking (even on non-numerical benchmarks) but also shed the light on two novel issues: prior array expansion may enable loop transformations that authorize profitable register promotion, and combination with alternative scheduling techniques (beyond interchange and fusion) may significantly improve performance in fine-tuning register usage and instruction-level parallelism.

Our main contributions are the following.

Register promotion. We revisit loop transformations for register promotion in a more general setting, building on the most advanced techniques in array data-flow analysis, array expansion, storage mapping optimization, and affine scheduling.

Storage mapping optimization. We extend a folding technique to better handle tiled iteration spaces and exploit the topmost level of the memory hierarchy. The main idea consists in capturing inter-iteration and cross-loop reuse patterns into so-called reuse buffers.

Computational biology. We present an optimized implementation of an approximate pattern matching algorithm for modern microprocessors. We experience strong speed-ups — over 6 on the Itanium1 and 5 on the Itanium2, with a number of instructions per cycle (IPC) close to optimal. This promises a significant impact on the relative merits and applicability ranges of pattern matching algorithms.

The paper is organized as follows: Section 2 discusses the optimization of a computational biology benchmark and further motivates our approach, Section 3 presents our algorithm for register promotion in the polytope model, before the conclusion.

2. BENCHMARK APPLICATION

We study the pattern matching algorithm BPR of practical use for computational biology (and some data mining applications). It is one of the main approximate string matching algorithms implemented in the NR-grep utility (non-deterministic reverse grep version 1.1.1) [35].

As opposed to well known tools like grep and agrep, NR-grep is based on the *bit-parallel* simulation of a non-deterministic automaton. Practically, it makes use of each individual bit in a 64 bit register to encode up to 64 states simultaneously. In addition, NR-grep implements *approximate* string matching algorithms, suitable for computational biology.

Practically, the implementation of BPR takes a (long) text and a length m pattern, and search for occurrences of the pattern with at most k errors (character insertion, deletion, substitution or swap). Useful values of k range from 0 (exact string matching) to $m/2$, and beyond for large alphabets. The main kernel of BPR is found in file `esimple.c`, function `esimpleScan` in the "forward" case with $k \geq 3$, see Figure 15.

Compared to the running example, BPR is not perfectly nested, uses one additional array `T` with similar access patterns, and implements a lot of bitwise logical operations. Interestingly, it also includes subscript of subscript references (to a read-only array) and uses irregular control structures to report matches (unpredictable early exit). None of these irregularities is a serious threat for array and loop transformations, but both of them suggest the application of modern techniques like *fuzzy* array data-flow analysis [14,

```
for (i=0; i<top; i++) {
  B1 = Bits[Text[i]];
  B2 = (B1 << 1) | ONE;
  oD = D[0];
  nD = D[0] = (oD << 1) | B1;
  for (j=1; j<=k; j++) {
    nD = ((D[j] << 1) | B1) & oD &
         ((oD & nD) << 1) & (T[j] | B2);
    T[j] = (oD << 2) | B1;
    oD = D[j]; D[j] = nD;
  }
  if (!(nD & E)) break;
}
```

Figure 15: Kernel of the BPR algorithm

47] and extensions of array expansion and storage mapping techniques to irregular nests [2, 13]. In addition, the unpredictable early exit requires *speculative* execution of the outer loop to implement unroll-and-jam or other scheduling transformations.

2.1 Manual Optimizations

The kernel in Figure 15 is considered as highly optimized by pattern matching experts: it is used for several empirical complexity evaluations [35]. A quick performance evaluation on all variants of the IA64 (Itanium) architecture confirms that the nest is *memory-bound*: locality is not an issue here, but performance suffers from compulsory misses on the text, and most of all, on L1 cache access bottlenecks.

Compulsory misses are easily handled by prefetching. Factoring loads to the text by (aligned) chunks of 8 characters helps solve the second issue, but the rate of loads and stores per iteration is still a major source of performance degradation. A more careful analysis shows that computations are grouped in *short dependent chains*, and that the number of arithmetic and logical operations is not sufficient to cover the L1 cache latency. This is obvious on the Itanium1 processor — 2 cycles latency — but also critical on the Itanium2 — even with a 1 cycle latency. Loop unrolling does increase instruction-level parallelism (ILP) and performance, but does not solve the memory bottleneck issue.

As a result, it appears that the obvious method to achieve significant performance gains is to increase register reuse. This implies tiling the loop nest and thus, expanding arrays `D` and `T`. Besides tiling, we wish to apply other loop transformations to interleave a sufficient number of short dependent chains, to improve ILP. Skewing is very helpful in this case: iterations spanning oblique fronts defined by `i+j` = *constant* can be executed in parallel. However, this transformation has an impact on locality and on the complexity of the generated code (array subscripts, loop bounds). Fortunately, the skewed version has a better intra-tile locality, leading to shorter life times after scalar promotion, avoiding unnessesary spill code for large tile sizes.

Overall, it is very hard to tell which version will have the best performance. Depending on whether skewing is applied or not, and whether it is applied before or after tiling, we must evaluate three candidate versions of the optimized code, namely `rect`, `skew` and `pllm`. Figure 16 shows the tile shapes and schedules for these three candidates. Each version is parameterized by the tile height — p — and width — q. Except the reuse buffer — the vertical bar depicted on the figures — *all* other accesses to arrays `D` and `T` have been promoted to scalars.

Suppose that a modern loop-restructuring compiler could discover these performance bottlenecks and decide on an appropriate transformation. Loop-carried dependences would still hamper register blocking, like in the running example. Skewing to improve ILP would also be impossible. The only way to apply the required transformations is to resort to array expansion (with array data-flow analysis), relying on a phase of storage mapping optimization to reduce the memory footprint and provide opportunities for register promotion.

2.2 Experiments

We used an Itanium2 1.3 GHz (Madison) workstation with HP's ZX1 chip-set and Intel's Electron compiler version 8. Measurements were conducted for several values of the pattern length m, alphabet size σ, and error k. We checked that the number of matches (early exists) never exceeded a few thousands.

First of all, it appears that `skew` and `pllm` — the second and third candidate version in Figure 16 — have very similar performances; we will thus limit ourselves to the results on `rect` and `skew`.

We consider two data-sets:

- `dna` corresponds to searches in the 600 kilobase genome of the *buchnera* bacteria ($\sigma = 4$) , with pattern size $m = 41$;
- `txt` corresponds to searches in a 10 megabyte English LaTeX document ($\sigma = 127$), with pattern size $m = 24$.

Speed-ups are measured with respect to the base version of NR-grep optimized with Electron (best optimization parameters with profiling), running for approximately 150 million cycles on `dna` (more than 100 milliseconds) and 1500 million cycles on `txt` (more than 1 second).

The tile's width and height are denoted by p and q, respectively. We first study the combined effect of all parameters except q.

Figure 17 shows the speed-ups of the two optimized versions (`rect` and `skew`), on the two data-sets (`dna` and `txt`), for $k = 13$ and $k = 19$ errors, for tile widths $p = 8$ and $p = 16$, and for the *best* tile height. Performance of both versions is very good in all cases — speed-up of 3 or more. Yet we experience significant variations, depending on the version, data-set and parameters.

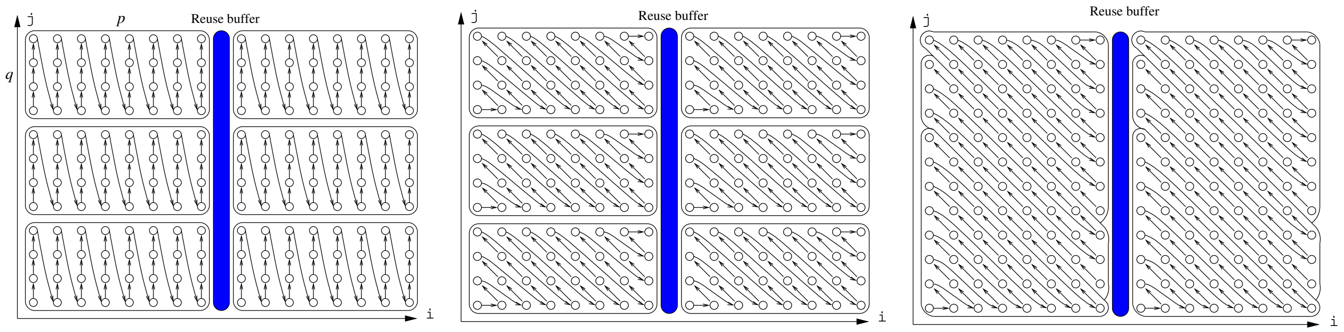
- The schedule of the nest is imporant: the skewed and tiled version is always better than tiling alone, up to 25.8% for `dna`, $k = 19$ and $p = 16$. This is due to a better exploitation of ILP without deteriorating locality.
- Increasing the tile width has a rather unpredictable impact. This is due to the large size of the loop body after full unrolling of the inner loops. Inspecting the generated code shows that the compiler sometimes produce spill code, having exhausted the 128 available registers.

dna	rect $k = 13$	skew $k = 13$	rect $k = 19$	skew $k = 19$
$p = 8$	2.99	3.31	4.60	5.08
$p = 16$	2.98	3.45	4.06	5.11

txt	rect $k = 13$	skew $k = 13$	rect $k = 19$	skew $k = 19$
$p = 8$	3.00	3.40	3.98	4.28
$p = 16$	2.99	3.54	3.41	4.05

Figure 17: Best speed-ups for $p = 8$ and $p = 16$

These results are confirmed by the IPC metric. On most combinations of parameters and versions, we get over 5 IPC, with a peak 5.6 on the best case, for an optimal value of 6 on the Itanium. However, for some cases, performance is sub-optimal; e.g., 3.79 IPC for $k = 19$, $p = 16$, version `rect` and data-set `txt`.



`rect`: tiling only. `skew`: tiling before skewing. `pllm`: skewing before tiling.

Figure 16: Three candidate schedules for the optimized BPR kernel

Let us now study the effects of the tile height (q). Figure 18 shows the speed-up for `dna`, $k = 19$, $p = 8$ and $p = 16$, varying the tile size q from 3 to 10. Beyond the performance advantage of `skew` with respect to `rect`, the results are hard to predict from a compiler perspective. Even this advantage comes down to a tiny difference for some cases, like $p = 8$ and $q = 5$. Most of these variations are due to the complex interplay of the tile shape with the relative size of the prelude/postlude of the tiled loops (when q does not divide $k+1$), and with the back-end optimizer of the Electron compiler (software-pipelining, register allocation, spilling heuristic).

These results confirm that predicting the interplay of loop transformations is a tough problem for modern compilers and architectures. We thus advocate for a flexible approach to register promotion that does not restrict the applicable loop transformations to register blocking alone. Any other framework would not be applicable in future optimizers that are likely to rely on feedback information and iterative (or adaptive) schemes [23, 15, 20].

To conclude this section, we showed that both array expansion and storage mapping optimization are necessary to optimize the memory-bound BPR kernel. Interestingly, we also showed that significant performance benefits may be obtained by combining this extended register blocking technique with other performance-increasing transformations, like skewing to increase ILP. The next section will take the point of view of compiler designers. It will describe how *this combination of transformations is made possible by the expressiveness of multidimensional affine schedules* [19].

3. POLYHEDRAL TRANSFORMATIONS

Our application to the BPR algorithm showed that dependence removal is not only important for parallelization (ILP in this case), but for register promotion as well. It enables locality-improving transformations and leads to strong speed-ups.

3.1 State-of-the-Art Array Folding

Let us summarize the storage mapping optimization method by Lefebvre and Feautrier [26]. In this presentation, we assume a *static-control* nest [17], i.e., affine bounds, conditional guards and subscripts with respect to surrounding loop counters and symbolic constants, but our method extends to irregular nests through the extended algorithms in [3, 13].

About the polytope model

We first recall classical results and definitions. Each iteration of a statement is called an *instance*. The instance of a statement s for an iteration vector v (the vector formed by the surrounding loop counters) is denoted by $\langle s, v\rangle$.

The polytope model assume that the relative ordering of every iteration of every statement is fully characterized through a collection of *affine schedules*. An affine schedule is a function from iteration vectors to *lexicographically ordered time vectors*. For a given statement s, the corresponding function θ_s is called the *affine schedule* of s [19]. This model can represent parallel and sequential schedules obtained through most loop transformations [22, 5].

Array data-flow analysis yields, for each reference r in right-hand side of a statement s, the function $\mathtt{RD}_{s,r}$ mapping any iteration of s to the precise instance that produced the value read through r (using the PIP tool [18]); it is a generalization of the classical *reaching definitions* [34] to polyhedral sets of instances and arrays. When a reference reads a value defined before the program fragment of interest, array data-flow analysis yields the special $\perp$ instance. In general, $\mathtt{RD}_{s,r}$ is a union of polyhedra and can be represented by a *quast*, short for *quasi-affine selection tree* [17] (a generalization of a last-write tree [32]). Each leaf in a quast is an *affine function* — characterized as a convex polyhedron — mapping a disjoint set of instances of s to their definitions. For example, if s is the second assignment in the running example, `p = D[j]`, and t is the fourth assignment, `D[j] = n`, then

$$\mathtt{RD}_{s,\mathtt{D[j]}}\big(\langle s,i,j\rangle\big) = \begin{cases} \text{if} \quad i=0 \quad \text{then} \quad \perp \\ \text{else} \quad \langle t, i-1, j\rangle \end{cases}$$

Indeed, reference `D[j]` in s reads the value produced by the last iteration of t, except for the first iteration of the outer loop where it reads the intial value of array `D`.

Array expansion is a direct application of array data-flow analysis [17]. Back to the running example, one may convert array `D` to the single-assignment array `E` in two simple steps:

- replace every reference to `D` in left-hand side by a reference to `E` subscripted with the surrounding loop indices,

  statement t becomes `E[i][j] = n;`
- replace every reference to `D` in right-hand side by a C implementation of the quast of its reaching definition, where $\perp$ corresponds to the original references to incoming values of `D`, and where non-$\perp$ leaves yield subscripts of `E`, statement s becomes `p = ` $\mathtt{RD}_{s,\mathtt{D[j]}}(\langle s,\mathtt{i},\mathtt{j}\rangle)$, i.e.,

  `p = (i==0) ? D[0] : E[i-1][j].`

This naive implementation of the right-hand side can be further optimized by polyhedral code generation techniques [41], hoisting all conditionals out of the inner loops. Modern code generation tools generate excellent control structures with no or very low overhead on medium-sized loop nests: for example, Bastoul's code generator [4] generate nearly-optimal conditionals and loops for hundreds of

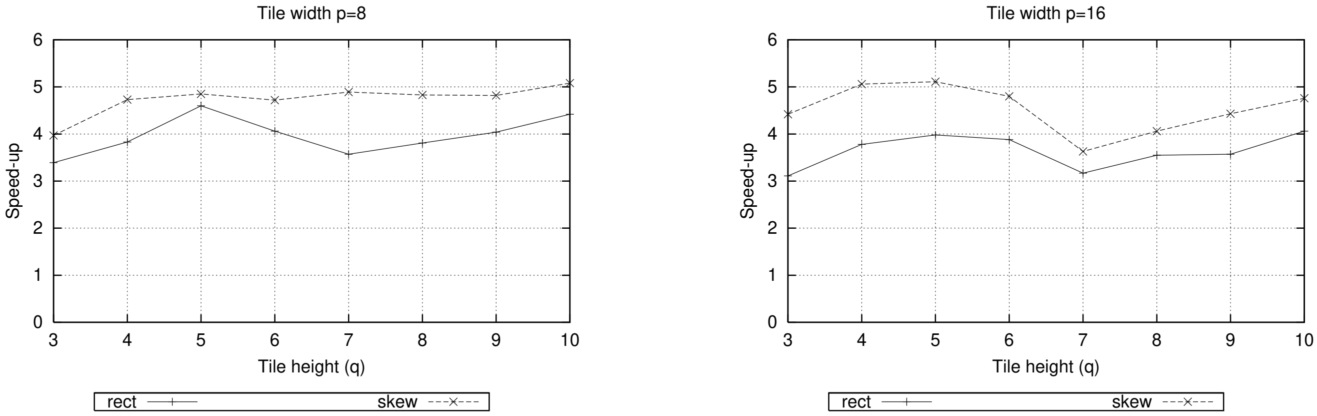


Figure 18: Tile height effect on performance

INTERFERENCE(s,s',A)

$d_1 \leftarrow \max(\text{depth}(s),\text{depth}(s'))$

For each (s'',r'') such that r'' is a reference to A in right-hand side, do

$d_2 \leftarrow \text{depth}(s'')$

Compute $\mathrm{RD}_{s'',r''}$

For each quast leaf a in $\mathrm{RD}_{s'',r''}$, do

For $p \in \{1,\ldots,d_1\}$ and $q \in \{1,\ldots,d_2\}$, do

$$\mathrm{I}^{a,s'',r''}_{p,q} \leftarrow \big\{(v,v') \mid v \in D_s \wedge v' \in D_{s'} \wedge v'' \in D_{s''} \wedge \theta_s(v) \ll_p \theta_{s'}(v') \wedge \theta_{s'}(v') \ll_q \theta s''(v'') \wedge v = \mathrm{RD}^a_{s'',r''}(v'')\big\}$$

Return $\bigcup_{p,q,a} \mathrm{I}^{a,s'',r''}_{p,q}$

Figure 19: Algorithm to compute the interference relation

MAXIMALREUSEDISTANCE(A)

For each (s,s') such that A is in left hand side of both s and s', do

For $k \in \{1,\ldots,\text{depth}(s)\}$, do

$\mathrm{WS}^k_{s,s'} \leftarrow \mathrm{INTERSECT}\big(\mathrm{INTERFERENCE}(s,s',\mathtt{A}), \ll_k\big)$

For $k \in \{1,\ldots,d\}$, do

For each (s,s') such that A is in left hand side of both s and s', do

$\Delta^k_{s,s'} \leftarrow \big\{(v'-v) \mid (v,v') \in \mathrm{WS}^k_{s,s'}\big\}$

$M^k_{s,s'} \leftarrow \max_{\ll_k} \Delta^k_{s,s'}$

$$m^k_{s,s'} \leftarrow \begin{cases} \infty & \text{if one of the leaves of the quast } M^k_{s,s'} \text{ is not a constant} \\ 0 & \text{if all leaves of the quast } M^k_{s,s'} \text{ are } \bot \\ \text{otherwise, the maximum value of the leaves in the quast } M^k_{s,s'} \end{cases}$$

$m^k_s \leftarrow \max_{s'} m^k_{s,s'}$

Figure 20: Algorithm to compute the maximal reuse distance

loop nests in the SpecFP and Perfect Club benchmarks, some with thousands lines of code. On the running example, it automatically peels the inner loop and generate the code in Figure 2.

Algorithm overview

For each pair of quast leaves, one may compute the relation between instances producing values that are simultaneously alive, according to the affine schedules (using the PolyLib [30]). This relation is called the *interference* relation, a generalization of the classical interference for register allocation [34] to polyhedral sets of instances and arrays.

The algorithm is formally stated in Figure 19. It uses the following additional definitions: depth(s) is the nesting depth of statement s, D_s is the iteration domain of s (the set of iteration vectors associated with executions of statement s), $\ll$ is the lexicographic ordering of vectors, $\ll_k$ is the lexicographic ordering at depth k ($u \ll_k v$ if u and v share a common prefix on the $k-1$ first dimensions and the k-th component of u is lower than the one of v), and $\mathrm{RD}^a_{s'',r''}$ is the affine function characterized by leaf a of the quast $\mathrm{RD}_{s'',r''}$.

Considering the loop nest in Figure 3, an affine schedule for the unique statement is

$$\theta(i,j) = (i,j).$$

From the lexicographic order, iteration (i,j) executes before $(i,j+1)$, which executes before $(i+1,j)$, etc. Considering the nest in Figure 4, affine schedules for the first and second statements are

$$\theta_1(i,j) = (i,2j) \text{ and } \theta_2(i,j) = (i,2j+1).$$

From the lexicographic order, iteration (i,j) of statement 1 executes before iteration (i,j) of statement 2, which itself executes before iteration $(i,j+1)$ of statement 1, etc. Thanks to the flexibility of affine schedules, we will show that storage mapping optimization smoothly handles inter-iteration and cross-loop reuse.

For each depth k of the loop nest, one may deduce a function WS^k_s mapping each instance i of a statement s to the *set of interfering writes following i* according to the affine schedule (using the PolyLib) — this set is called the *working set* of i. One may then compute the *maximal reuse distance* m^k_s between each instance of s and its last interfering write *at depth k* (using PIP). It was proven in [26] that a single-assignment array subscript `[i_1]...[i_d]` in left hand-side of s may be safely replaced by

$$\mathtt{[i_1\%}(1+m^1_s)\mathtt{]}\ldots\mathtt{[i_d\%}(1+m^d_s)\mathtt{]}.$$

The algorithm is formally stated in Figure 20. It uses the following additional definitions: d is the depth of the full nest and INTERSECT is the intersection of polyhedra. Both INTERSECT and the computation of $\Delta^p_{s,s'}$ correspond to PolyLib functions.

In a final step, each array in left-hand side of a statement is given a unique name, and the effects of these transformations are propagated to the uses in right-hand side. In addition, non-convex array access patterns (lattice footprints) with strided rows or columns of useless data (never read nor written) can be further contracted by dividing the corresponding subscripts by the strides greatest common divisor [40]. We applied optimization to array `B` in Figure 11.

Application to the running example

Let us now informally apply this algorithm to the code in Figure 5. The first statement stores a value in `E[i][j]` that does not escape the loop body. The second and fourth statements record in `E[i+1][j]` and `E[i+1][j+1]` a value that is later consumed by the next iteration of the outer loop. The third statement records in `E[i][j+1]` a value that is later consumed by the next iteration of the inner loop. No value flows across more than one iteration of the outer loop: the maximal distance at depth 1 is 1. With a fixed value of `i`, some values (produced by the second and fourth statement) flow across all iterations the inner loop: the maximal distance at depth 2 is k, hence $m_1^2 = m_2^2 = m_3^2 = m_4^2 = \infty$, i.e., there is no folding opportunity. One may thus fold array `E` down to two columns: `E[i][j]` can be replaced by `E[i%2][j]`, which is equivalent to `E[1][j]` since `i` is an odd number. No renaming is necessary here since `E` is a temporary single-assignment array.

We have shown that array contraction cannot remove any dimension after conversion to single-assignment form and unroll-and-jam. Storage mapping optimization does a better job: the folded array is only twice the size of array `D` in the original nest.

Important remarks

Interestingly, the *number of folded dimensions* of the resulting arrays is proven maximal for all affine schedules and storage mappings based on dimension-per-dimension foldings [40]. This strong result provides a "no-harm" guarantee for conversion to single-assignment form: if no schedule transformation is applied (fusion, interchange, etc.), it proves that the folded arrays will be no larger than the ones before expansion. Of course, loop transformations may modify the liveness of values, but this optimality result is a good reason to expect that the scalar promotion benefits will not be counterbalanced by side-effects on the memory footprint. Nevertheless, we cannot prove that the *size of folded dimensions* will be minimal. Indeed, in addition to improving the opportunities for scalar promotion, our improved technique reduces the size of temporary arrays by a factor of two (compared to the method by Lefebvre and Feautrier) on the skewed version of the BPR algorithm.

Eventually, this algorithm assumes a known schedule; but some approaches to storage mapping optimization are schedule independent — including [44] and one of the methods in [45]. This allows a greater flexibility in many parallelization purposes, and for some cases of sequential program transformations as well. However, it would not be sufficient to fold array `E` in our example: indeed, loop interchange is a legal transformation in the single-assignment version but does not support any folding of the first dimension of `E`.

3.2 Extended Register Blocking Algorithm

Manual optimization of the running example showed that array folding opportunities may be increased thanks to intermediate reuse buffers. Our extended algorithm combines array expansion, tiling, insertion of reuse buffers, and storage mapping optimization.

First of all, we define a *hierarchical boundary* between the inner depths where we will perform full unrolling for register promotion, from the outer depths of the loop nest. Typically, such a separation would be the natural result of a multidimensional tiling transformation [11]: we would not explicitly unroll the inner loops until the last optimization phase (scalar promotion), but we would clearly mark the hierarchy between the "array" and "scalar" reigns.

Notice that an aggressive but partial unrolling may be detrimental, because no reuse buffers have been introduced at the innermost levels to capture inter-iteration reuse. If full unrolling is too expensive (in terms of code size and instruction caches), it just means that the boundary is not correctly defined or that the tiles are too large.

Reuse buffers

Reuse buffers are closely related to the generation of communications in a data-parallel language like HPF [37]. For a given statement s at depth d in a loop nest, and for a given depth $k \le d$, we automatically insert a buffer to decouple inter-iteration reuse from intra-iteration reuse at depth k. If the left-hand side of s is of the form $\mathtt{A[i_1]...[i_d]}$, and *if and only if*, some values defined by s flow to other iterations of the surrounding loops, we insert a new statement s' right before/after s replicating the stored value in a new buffer $\mathtt{A_{buffer}}$. Practically, we create a statement

$$s': \quad \mathtt{A_{buffer}[i_1]...[i_d] = A[i_1]...[i_d]},$$

with the same schedule as s and whose iteration domain is restricted to the set of iteration vectors v of s such that there exists an iteration vector v' of a statement s' with a reference r' in right-hand side such that $\mathrm{RD}_{s',r'}(v') = v$ and $v \ll_k v'$. Thanks to the result of array dataflow analysis again, it is easy to propagate the new buffer in *all* references accessing values reused over iterations at depth k. This process may be repeated for all statements (or only those of interest to inter-iteration scalar promotion). In the resulting nest, we have transfered to the separate new buffers, all references reading values produced at a different iteration of a surrounding loop at depth k.

Applying this technique to the outer and inner loops of the running example — respectively at depth 1 and 2 — produces the code in Figures 9 and 10, introducing the new arrays `B` and `C`.

Notice these reuse buffers are just a convenient way to implement index-set splitting: they easily separate iterations producing values that "escape" to other iterations, from the ones producing local values only.

Folding heuristic

Despite the optimality result and alternative strategies proposed in [40], there is no reason for choosing a computation order or another for the maximal reuse distances at depth k (m_s^k). Both [26] and [40] make arbitrary choices, like folding array dimensions from the outermost loop inwards.

In the context of register promotion and reuse buffers, the situation is different. Recalling the *hierarchical boundary* concept introduced in this section, we may safely begin the computation of m_s^k from the inner loops, taking the risk of generating a little more variable names in the innermost loop body. This is very acceptable, assuming that a good register allocation algorithm is implemented in our compiler back-end (to carefully trade register pressure for ILP). The evaluation of m_s^k for the outer loops may lead to smaller values since the working sets have been restricted to fixed values of the inner loop counters. Practically, we thus reverse the ordering proposed by [26]: we compute m_s^k from the innermost level outwards.

This optimization is responsible for the reduction of the memory footprint in the `pllm` version of BPR: the reuse buffer (across the outer loop) has half the storage size of the two other versions, i.e., exactly the same size as arrays `D` and `T`. This is due to subtle interactions between the affine schedule and the liveness of values stored in the reuse buffer.

Algorithm summary

It is not the purpose of this paper to discuss locality-improving schedule transformations. We will thus assume that an arbitrary scheduling algorithm is applied to the single-assignment loop nest. In the case of the running example, we would reproduce the manual transformations through two-dimensional tiling. For the application to NR-grep, this scheduling algorithm would optionally combine two-dimensional tiling with a skewing transformation, either before or after tiling, to reproduce the three candidate versions presented in Section 2.1.

For complexity reasons, it is much better not to unroll the loops until the final scalar promotion phase. Indeed, storage mapping optimization is an expensive process (most polyhedral operations are exponential in the worst case). We thus prefer tiling to unroll-and-jam to improve locality.

Putting it altogether, the main steps of our extended algorithm are as follows.

1. Run array data-flow analysis for every right-hand side array reference.
2. Convert all temporary arrays and scalars (not escaping the nest) to single-assignment form. Alternatively, forward substitution may be used to remove memory-based dependences on scalars, reducing the memory footprint of the expanded program. On demand, one may expand some escaping variables as well by insertion of copy-in/copy-out code.
3. Apply any affine scheduling algorithm, typically a locality-improving one like hierarchical tiling. This scheduling phase may include other loop transformations to address performance issues like instruction and thread level parallelism.
4. Fix a hierarchical boundary, identifying the aggressively unrolled inner loops where scalar promotion will occur. This boundary would typically correspond to the tile dimension, i.e., depth 3 in the tiled running example.
5. Insert reuse buffers at all depths below (enclosing) the hierarchical boundary to capture values flowing across iterations of the outermost loops (not only the innermost one).
6. Apply storage mapping optimization from the inner loops outwards.
7. Fully unroll the inner loops, up to the hierarchical boundary.
8. Apply a classical scalar promotion algorithm to eliminate as many array accesses as possible.
9. Apply forward substitution to remove scalar duplicates. This may require further unrolling of the inner loops or make use of architectural features like rotating register files.

4. CONCLUSION AND FUTURE WORKS

We revisited loop transformations for register promotion in the more general setting of polyhedral loop transformations, array expansion and storage mapping optimization. This is motivated by a detailed study of the limitations of classical approaches — including scalar reuse across outer-loop iterations and array folding — and by empirical evidence on a real-world computational biology example. Besides this compilation issue, our strong speed-ups on this kernel modify the fragile balance between the relative merits of pattern matching algorithms [35].

On the algorithmic side, we combine array expansion with other enabling transformations to improve the opportunities for register promotion. We also extend a schedule-dependent storage mapping optimization technique to better handle tiled iteration spaces and explicitly manage inter-iteration reuse through specific buffers. Finally, we sketched a flexible register blocking algorithm that integrates these techniques, without restraining the application of loop transformations targeting other architectural components (caches, predictors, ILP, threads, etc.). We manually applied this algorithm to reproduce the speed-up results on the computational biology benchmark.

Some of the cited storage mapping optimization methods were implemented by their authors, but only applied to small kernels. Indeed, no large-scale implementation of polyhedral techniques for locality improvement have been ever released. This is one of the major goals of a more extensive project in our team: a framework and polyhedral transformation library for iterative and feedback-directed optimization of loop nests [5]. Register blocking is one of the obvious optimizations that such a framework should provide. Implementing a robust array data-flow analysis is the first priority, array expansion, insertion of reuse buffers and storage mapping optimization will follow.

Our work may have applications beyond register promotion. Our contributions should naturally apply to optimizations for random-access memory structures as well, especially those where explicit data transfers are either required (like for register-to-memory transfers) or highly beneficial: scratch-pad or local memories, translation buffers, non uniform shared memories, and disks.

5. REFERENCES

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan and Kaufman, 2002.

[2] D. Barthou, A. Cohen, and J.-F. Collard. Maximal static expansion. In *25th ACM Symp. on Principles of Programming Languages*, pages 98–106, San Diego, California, Jan. 1998.

[3] D. Barthou, J.-F. Collard, and P. Feautrier. Fuzzy array dataflow analysis. *J. of Parallel and Distributed Computing*, 40:210–226, 1997.

[4] C. Bastoul. Efficient code generation for automatic parallelization and optimization. In *ISPDC'2 IEEE International Symposium on Parallel and Distributed Computing*, Ljubjana, Slovenia, Oct. 2003.

[5] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting polyhedral loop transformations to work. In *Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*, LNCS, College Station, Texas, Oct. 2003.

[6] C. Bastoul and P. Feautrier. Improving data locality by chunking. In *CC'12 Intl. Conference on Compiler Construction, LNCS 2622*, pages 320–335, Warsaw, Poland, april 2003.

[7] R. Bodík, R. Gupta, and M. L. Soffa. Load reuse analysis: Design and evaluation. In *ACM Symp. on Programming Language Design and Implementation (PLDI'99)*, Atlanta, Georgia, May 1999.

[8] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *ACM Symp. on Programming Language Design and Implementation (PLDI'90)*, White Plains, New York, June 1990.

[9] P.-Y. Calland, A. Darte, Y. Robert, and F. Vivien. Plugging anti and output dependence removal techniques into loop parallelization algorithms. *Parallel Computing*, 23(1–2):251–266, 1997.

[10] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating point operations in loops. *ACM Trans. on Programming Languages and Systems*, 16(6), Nov. 1994.

[11] L. Carter, J. Ferrante, and S. F. Hummel. Efficient multiprocessor parallelism via hierarchical tiling. In *SIAM Conference on Parallel Processing for Scientific Computing*, Feb. 1995.

[12] A. Cohen, S. Girbal, and O. Temam. Facilitating the exploration of compositions of program transformations. Research report 5114, INRIA Futurs, France, Feb. 2004.

[13] A. Cohen and V. Lefebvre. Optimization of storage mappings for parallel programs. In *EuroPar'99*, number 1685 in LNCS, pages 375–382, Toulouse, France, Sept. 1999. Springer-Verlag.

[14] J.-F. Collard, D. Barthou, and P. Feautrier. Fuzzy array dataflow analysis. In *ACM Symp. on Principles and Practice of Parallel Programming*, pages 92–102, Santa Barbara, California, July 1995.

[15] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *J. of Supercomputing*, 2002.

[16] E. Duesterwald, R. Gupta, and M. L. Soffa. A practical data flow framework for array reference analysis and its use in optimization. In *ACM Symp. on Programming Language Design and Implementation (PLDI'93)*, pages 68–77, Albuquerque, New Mexico, jun 1993.

[17] P. Feautrier. Array expansion. In *ACM Int. Conf. on Supercomputing*, pages 429–441, St. Malo, France, July 1988.

[18] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, Sept. 1988.

[19] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II, multidimensional time. *Int. J. of Parallel Programming*, 21(6):389–420, Dec. 1992. See also Part I, one dimensional time, 21(5):315–348.

[20] G. Fursin, M. O'Boyle, and P. Knijnenburg. Evaluating iterative compilation. In *11th Workshop on Languages and Compilers for Parallel Computing*, LNCS, Washington DC, July 2002. Springer-Verlag.

[21] M. Jiménez, J. Llabería, and A. Fernández. Register tiling in nonrectangular iteration spaces. *ACM Trans. on Programming Languages and Systems*, 24(4):409–453, 2002.

[22] W. Kelly. Optimization within a unified transformation framework. Technical Report CS-TR-3725, University of Maryland, 1996.

[23] T. Kisuki, P. Knijnenburg, K. Gallivan, and M. O'Boyle. The effect of cache models on iterative compilation for combined tiling and unrolling. In *Parallel Architectures and Compilation Techniques (PACT'00)*. IEEE Computer Society Press, Oct. 2001.

[24] K. Knobe and V. Sarkar. Array SSA form and its use in parallelization. In *25th ACM Symp. on Principles of Programming Languages*, pages 107–120, San Diego, California, Jan. 1998.

[25] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *ACM Symp. on Programming Language Design and Implementation (PLDI'97)*, pages 346–357, Las Vegas, Nevada, June 1997.

[26] V. Lefebvre and P. Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24(3):649–671, 1998.

[27] A. W. Lim and M. S. Lam. Communication-free parallelization via affine transformations. In *24th ACM Symp. on Principles of Programming Languages*, pages 201–214, Paris, France, jan 1997.

[28] A. W. Lim, S.-W. Liao, and M. S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *ACM Symp. on Principles and Practice of Parallel Programming (PPoPP'01*, pages 102–112, 2001.

[29] V. Loechner, B. Meister, and P. Clauss. Precise data locality optimization of nested loops. *J. of Supercomputing*, 21(1):37–76, Jan. 2002.

[30] V. Loechner and D. Wilde. Parameterized polyhedra and their vertices. *Int. J. of Parallel Programming*, 25(6), Dec. 1997. `http://icps.u-strasbg.fr/PolyLib`.

[31] J. Lu and K. Cooper. Register promotion in C programs. In *ACM Symp. on Programming Language Design and Implementation (PLDI'97)*, pages 308–319, June 1997.

[32] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array dataflow analysis and its use in array privatization. In *20th ACM Symp. on Principles of Programming Languages*, pages 2–15, Charleston, South Carolina, Jan. 1993.

[33] K. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, july 1996.

[34] S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, 1997.

[35] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings*. Cambridge University Press, 2002.

[36] D. Parello, O. Temam, and J.-M. Verdun. On increasing architecture awareness in program optimizations to bridge the gap between peak and sustained processor performance? matrix-multiply revisited. In *SuperComputing'02*, Baltimore, Maryland, Nov. 2002.

[37] G.-R. Perrin and A. Darte, editors. *The Data Parallel Programming Model*. Number 1132 in LNCS. Springer-Verlag, 1996.

[38] G. Pike. *Reordering and Storage Optimizations for Scientific Programs*. PhD thesis, University of California Berkeley, 2002.

[39] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):27–47, Aug. 1992.

[40] F. Quilleré and S. Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Trans. on Programming Languages and Systems*, 22(5):773–815, Sept. 2000.

[41] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *Intl. J. of Parallel Programming*, 28(5):469–498, Oct. 2000.

[42] R. Schreiber, S. Aditya, B. Rau, V. Kathail, S. Mahlke, S. Abraham, and G. Snider. High-level synthesis of nonprogrammable hardware accelerators. Technical report, Hewlett-Packard, May 2000.

[43] B. So, M. W. Hall, and P. Diniz. A compiler approach to design space exploration in fpga-based systems. In *ACM Symp. on Programming Language Design and Implementation (PLDI'02)*, June 2002.

[44] M. M. Strout, L. Carter, J. Ferrante, and B. Simon. Schedule-independant storage mapping for loops. In *ACM Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*, 8, 1998.

[45] W. Thies, F. Vivien, J. Sheldon, and S. Amarasinghe. A unified framework for schedule and storage optimization. In *ACM Symp. on Programming Language Design and Implementation (PLDI'01)*, pages 232–242, 2001.

[46] P. Tu and D. Padua. Automatic array privatization. In *6th Workshop on Languages and Compilers for Parallel Computing*, number 768 in LNCS, pages 500–521, Portland, Oregon, Aug. 1993.

[47] D. Wonnacott and W. Pugh. Nonlinear array dependence analysis. In *Proc. Third Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers*, 1995. Troy, New York.